EXPRESS MAIL LABEL NO. EV339203824US
DATE OF DEPOSIT: June 26, 2003

AN INTERMEDIATE REPRESENTATION FOR MULTIPLE EXCEPTION HANDLING MODELS

TECHNICAL FIELD

The technical field relates to components of a compiler computer program. More specifically the field relates to an intermediate representation of exception handling constructs for compiling a program.

BACKGROUND

Generally speaking, a translator is a computer program that receives as its input a program written in one computer programming language and produces as its output a program in another programming language. Translators that receive as their input a high-level source language (e.g., C++, JAVA, etc.) and generate as their output a low-level language such as assembly language or machine language sometimes are more specifically referred to as compilers. The process of translation within a compiler program generally consists of multiple phases. FIG. 1 illustrates a flow chart showing one such break down of the multiple phases of a compiler. The source program representation in source code is received at 110. Then at 120 the lexical analyzer separates the characters of the source code into logical groups referred to as tokens. The tokens may be key words in the syntax of the source language such as, IF or WHILE, operators such as, + and – , identifiers and punctuation symbols. At 130, the syntax analyzer groups the tokens together into syntactic structures such an expression or a statement. At 140, an intermediate representation (IR) of the source code, including the exception handling constructs, is generated to facilitate compiler back end operations such as code optimization at 150 and then code generation at 160. There can be multiple intermediate representations within a compiler process. During the code optimization phase 150 various techniques may be directed to improving the intermediate representation generated at 140 so that the ultimate object code runs faster

and uses less memory. During the final phase at 160, the code generator produces the target program (object code) 170 to be executed by a processor.

Exception handling is invoked when a flaw in the source program is detected. In the existing compiler frameworks, exception handling constructs within the source program are processed separate from the main control flow of the intermediate representation. Traditionally, exception handling constructs are not explicitly represented in the control flow of the intermediate representation. In one well known technique, regions within the source code where exception handling constructs are detected are delimited from the main control flow and thus not subject to the same code optimization techniques as the main control flow. In yet another method, the exception handling constructs are captured within a table outside of the main control flow and the compiler back end processes them separately. Thus, there is a need for intermediate representation for exception handling constructs that allows such constructs to be explicitly represented within the main control flow to take advantage of the same code optimizations and code generation techniques (i.e., compiler back end) as the rest of the source code.

Also, traditionally, intermediate representations have been specific to a source language. Thus, compilers have to be aware of the specific exception handling models of the source language associated with each representation. For our purposes, these exception handling models can be typically characterized by four features. The first feature determines if the exception is synchronous or asynchronous. A synchronous exception is associated with the action of the thread of control that throws and handles it. In this situation, an exception is always associated with an instruction of the thread. In other words, an exception handling action is invoked by an instruction when some condition fails. However, an asynchronous exception is injected into a thread of control other than thread that may have thrown and handled it. In Microsoft CLR, this may be caused by aborting a thread via a system API. Such exceptions are not associated to a

particular instruction. The effect is to raise an exception in the thread at some suitable point called a synchronization point.

Second, an exception may either terminate or resume the exception causing instruction. In the case of a terminating exception the instruction is terminated and a filter, handler, or a finalization action is initiated. However in the case of a resumption model the offending instruction can be automatically resumed after some handling action is performed. The Structured Exception Handling (SEH) constructs in C/C++ fall into this category. This requires, typically, that the entire region including the exception causing instruction be guarded as if all memory accesses act like volatile accesses. Thus, disallowing any optimization of the memory accesses.

Third, an exception handling model may be precise or imprecise. In precise exception handling models relative ordering of two instructions needs to preserve observable behavior of memory state. This means that a reordering of instructions cannot be performed if a handler or another fragment of code will see different values of variables. Languages such as C#, Microsoft CLR and C++ require a precise mechanism. In such models, the compiler may need to reorder exception instructions relative to each other and any other instruction whose effect is visible globally. In imprecise models, the relative order of instructions on exception effect is undefined and a compiler is free to reorder such instructions. In either model, the order between exception instructions and their handlers is always defined and is based on control dependencies. Some languages like Ada have an imprecise exception model.

Fourth feature of an exception handling model is how handler association is performed in various exception handling models. In most languages, including C++, C#, and Microsoft CLR, handler association is lexical and performed statically. This means that it is statically possible to identify the start of the handler code and this is unique. As explained below this attribute of statically identifying handler bodies may be used to generate the intermediate representation of the exception handling instructions. Thus, there is a need for a single uniform framework for intermediately

representing exception handling constructs that is uniform across multiple models for representing exception handling and is capable of accounting for the various attributes of such models described above.

SUMMARY

As described herein, a uniform intermediate representation of exception handling constructs may be used for expressing exception handling models of various languages. In one aspect, a single set of instructions related to the intermediate representation are described herein for expressing multiple different exception handling mechanisms. For example, a common set of related instructions may be used to describe the control flow from a try region to a finally region and then to outside of the finally region. In yet another aspect, control flow from a try region to a catch region may be expressed using a common set of related instructions. Furthermore, filters guarding the handler or catch region may also be expressed. Control flow from a try region to the "except" region to pass the control back to the exception causing region under certain conditions may also be expressed. Exception handling control flow related to object destructors may also be expressed using the uniform intermediate representation of the exception handling constructs.

In a further aspect, methods and systems are described herein for generating the uniform intermediate representation for expressing control flow of exception handling constructs. In one aspect, the intermediate representation may be generated by translating an intermediate language representation of the source code file. Multiple different intermediate languages may be used to generate the intermediate representation of exception handling constructs. In a further aspect, the intermediate representation of the exception handling constructs may be used by software development tools for such tasks as code generation, code optimization, analysis etc.

Additional features and advantages will be made apparent from the following detailed description of illustrated embodiments, which proceeds with reference to accompanying drawings.

BRIEF DESCRIPTION OF DRAWINGS

FIG. 1 is a flow chart representing the various processing phases of a typical compiler and its components.

FIG. 2 is a block diagram illustrating a system for generating an intermediate representation of exception handling instructions using a uniform exception handling framework capable of representing multiple language specific exception handling models.

FIG. 3A is a flowchart illustrating a method for generating an intermediate representation of exception handling instructions using a uniform exception handling framework capable of representing multiple language specific exception handling models.

FIG. 3B is a flowchart illustrating a method for reading an intermediate representation of software and generating an executable version therefrom.

FIG. 4 is a block diagram of one embodiment of the system of FIG. 2 showing a multiple IL readers for CIL and MSIL languages.

FIG. 5 is a diagram of one embodiment of a data structure for instructions in an intermediate representation.

FIG. 6 is a listing of a pseudo code representation of unguarded exception causing instructions.

FIG. 7 is a listing of an intermediate representation of the code of FIG. 6.

FIG. 8 is a listing of a pseudo code representation of a try code section with non-exception causing instructions guarded by a finally block.

FIG. 9 is a listing of an intermediate representation of the code of FIG. 8.

FIG. 10 is a listing of a pseudo code representation of a try code section with exception causing instructions guarded by a finally block.

FIG. 11 is a listing of an intermediate representation of the code of FIG. 10 along with the appropriate handler labels.

FIG. 12 is a listing of a pseudo code representation of a try code section with exception causing instructions guarded by two filters and two catch blocks.

FIG. 13 is a listing of an intermediate representation of the code of FIG. 12 along with the appropriate handler labels and filters related to the catch blocks.

FIG. 14 is a listing of a pseudo code representation of a try code section with exception causing instructions guarded by two filters, two catch blocks and a finalization code block.

FIG. 15 is a listing of an intermediate representation of the code of FIG. 14 along with the appropriate handler labels and filters related to the catch and finalization blocks.

FIG. 16 is a listing of a pseudo code representation of a nested try code section guarded by a catch block.

FIG. 17 is a listing of an intermediate representation of the code of FIG. 16 along with the appropriate handler labels and filters related to the nested and the outer catch and finalization blocks.

FIG. 18 is a block diagram illustrating one method for translating exception handling constructs from an intermediate language to another intermediate representation.

FIG. 19A is an illustration of a data structure of exception handling data table.

FIG. 19B is one illustration of a label map for mapping offsets to their labels.

FIG. 19C is another illustration of a label map after protected blocks are mapped to their respective offsets.

FIG. 20 is a flow chart of one method for using the exception handling data table and the containment information between protected blocks and their handlers and destination blocks for generating an intermediate representation.

FIG. 21 is a diagram showing one example of a range tree map for determining the containment relationship between the protected blocks and their handlers and destination blocks.

FIG. 22 is a listing of a C++ program illustrating construction and destruction of local objects.

FIG. 23 is a listing of a pseudo code representation for expressing the possible exception handling paths during the constructions and destruction of objects.

FIG. 24 is a listing of an intermediate representation of the code of FIGS. 22 and 23.

FIG. 25 is a listing of a C++ program illustrating conditional construction of expression temporary objects.

FIG. 26 is a listing of a pseudo representation for expressing the possible exception paths for conditional construction and destruction of expression temporary objects.

FIG. 27 is a listing of an intermediate representation of the code of FIG. 26.

FIG. 28 is a listing of a C++ program that returns an object by value.

FIG. 29A is a listing of an intermediate representation of the possible exception paths for destruction of objects by value shown in FIG. 28.

FIG. 29B is a continuation of the listing of FIG. 29A.

FIG. 30 is a listing of a C++ program throwing an object by value.

FIG. 31 is a listing of a pseudo code representation expressing the possible exception paths of throwing value type objects shown in FIG. 30.

FIG. 32 is a listing of an intermediate representation of FIG. 31.

FIG. 33 is a listing of a pseudo code representation of a try code section guarded by an except code block.

FIG. 34 is a listing of an intermediate representation of the code of FIG. 33.

FIG. 35 is a flow chart of an exemplary method for translating an intermediate language expressed in post fix notation form to another intermediate representation.

FIG. 36 is a diagram of one implementation of data structures to build an intermediate representation from reading code expressed in a post fix notation form.

FIG. 37 is a flow chart showing an exemplary method for using the data structures of FIG. 36 to build an intermediate representation by reading code expressed in post fix notation form.

FIG. 38A is a listing of an exemplary code section of FIG. 22 implemented using postfix notation.

FIG. 38B is the continuation of FIG. 38A.

FIG. 38C is the further continuation of FIGS. 38A and B.

FIG. 39 is a block diagram illustrating the state of data structures of FIG. 36 during the translation of the code of FIG. 38 to an intermediate representation.

DETAILED DESCRIPTION

Language independent intermediate representation of exception handling constructs

FIG. 2 illustrates a system 200 for implementing a uniform exception handling intermediate representation 230 for multiple source languages (205-208) for code optimization by the compiler back end 240. As shown in FIG. 2, the system 200 includes a intermediate language (IL) representation 210-213 for each of the multiple source code representations 205-208 which is parsed or read by an IL reader 220 which translates the multiple IL representations 210-213 to a single intermediate representation 230. The IL representation is a higher-level intermediate representation than the intermediate representation 230 and may be expressed in any number of well known intermediate languages such as MSIL (Microsoft CLR) (for C#, Visual Basic, JScript, C, and FORTRAN) and CIL (for C++). Even though the system 200 for generating a uniform exception handling framework for multiple languages is shown as having a single IL reader process for multiple source languages, it is possible to implement multiple such readers, each corresponding to one or more of the IL representations 210-213.

FIG. 3A illustrates a general overall method for using the IL reader 220 to generate a uniform set of intermediate representations for exception handling constructs expressed in a number of different source languages. At 310, the intermediate language representation of software (e.g., an intermediate language representation of a source code file) is received by the reader 220 and at 315, the file is read or parsed to identify exception handling constructs within the IL code stream (320). Then at 330, the reader 220 (which can also be thought of as a virtual machine) generates a single uniform intermediate representation of the exception handling constructs identified previously at 320. Such an exception handling frame work can then be used to simplify the processes of a compiler back end such as code optimizations and code generation.

The uniform intermediate representation of the software having the exception handling constructs can explicitly express exception handling control of the software. FIG. 3B shows a method 350 for generating executable from the uniform intermediate representation of the software. Such a method can be used, for example, by a compiler or other software development tool when generating an executable version (e.g., machine-specific code or other object code) for the software.

At 360, the uniform intermediate representation is read (e.g., by a compiler or other software development tool). For example, the uniform intermediate representation generated by the method of FIG. 3A can be used. Other transformations, translations, or optimizations to the uniform intermediate representation can be performed as desired.

At 370, a computer-executable version of the software is generated (e.g., by the compiler or other software development tool). The computer-executable version of the software implements the exception handling control flow of the software, based on the uniform intermediate representation.

FIG. 4 illustrates another embodiment of a system for generating a simple and uniform intermediate representation of exception handling constructs within multiple source languages expressed in form of multiple IL representations. As shown in FIG. 4, the source language group 410 supported within Microsoft's .NET framework (e.g., C#, C, Microsoft Visual Basic, Jscript, and FORTRAN) are first translated to a MSIL representation 440. However, because of its differences with other source languages C++ is expressed in another intermediate language known as CIL 430. The control flow and the exception handling models within the CIL and MSIL are expressed in fundamentally different ways and thus it may be necessary to provide separate IL readers (435 and 445) for CIL and MSIL representations.

Both the readers 435 and 445 may use appropriate algorithms implemented within their respective readers to parse or read their respective intermediate language code streams to express the exception handling constructs or instructions or expressions

within the intermediate language code stream using a uniform framework of exception handling instructions 450 to be provided to the back end 460. Part of the rest of this document below describes various components of such a language independent exception handling instruction set. Furthermore, examples of exception handling constructs within the intermediate language are shown translated to their respective language independent intermediate representations. The document also describes algorithms and methods for parsing the intermediate language and generating the intermediate representations of exception handling constructs.

Exception causing instructions explicitly expressed within the main control flow of the intermediate representation

Exception causing instructions are guarded by their handlers or finally regions. When an instruction causes an exception the control flow may pass to a handler and sometimes the handler may be conditionally selected based on the processing of filter instructions. Control may flow to finally regions of code based on exceptions or directly, either way, it will be processed and used to implement clean –up code. Finally, regions are always executed before the control is exited out of the corresponding try region. This mechanism can be used for implementing clean up code, such as closing of file handles, sockets, locks, etc. FIGS. 8 and 12 illustrate pseudo code representing various exception handling related instructions. FIG. 12 for example shows a try region guarded by two catch blocks. The choice of which of the two catch blocks (in FIG. 12) is to be processed is dependent on the results of processing a filter block. As a further example, FIG. 8 shows a try region guarded by a finally block.

As described with reference to FIG. 3, the intermediate language representations with various models for expressing exception handling maybe analyzed to determine control flow between the exception causing instructions and their respective handlers and continuations, which may then be explicitly expressed within the same control flow as the rest of the instructions that do not cause exceptions. One way to accomplish this

is to build a control flow representation using instructions with a modest memory allocation cost such as, one word per instruction. The handlers may be represented by instructions that use an exception variable which may be defined by an exception causing instruction. The handler or filter instructions can then test the exception variable and branch to the handler body or to another handler based on the value or type of the exception object. Similarly, instructions guarded by a finally clause in C++ or C# have control flow edges or pointers to instructions that capture the continuation for the target of the control transfer out of the finally region. The end of a finally region in this case may be modeled by an instruction that transfers control to the captured continuation at the start of a finally region. These features of the intermediate representation will be described in further detail below with reference to examples.

Format for Instructions

As noted above, the intermediate representation of exception handling constructs in the intermediate language representation may be expressed at an instruction level. FIG. 5 shows one such general implementation of a data structure for instructions or nodes (IR nodes) that will allow the exception handling constructs to be expressed within the control flow of the intermediate representation of the rest of the code. Specific intermediate representations of exception handling instructions and their functionality is described later in the document. Generally, IR instructions may be executable at various levels within a compiler component hierarchy. They have an operator (op-code) field and a set of source (or input) operands, a set of destination (or output) operands. These operands are typically references to symbol nodes. In addition, each of the source and destination operands may be typed and the operator and the types of the operand may be used to resolve any ambiguity. In the example instruction of FIG. 5, the operator at 504 has two source operands 506 and 507 and two destination operands 508 and 509.

The exception handling semantics may be represented by providing, each instruction 505 with a handler field 510 that points to a label instruction 520 which is the start of the handler 530 for that instruction 505. If the instruction cannot throw an exception then the handler field 510 of the instruction is set to NULL. If the instruction can throw an exception but has no handler then the compiler may build a special handler to propagate control out of the current method.

A textual notation for describing the IR instruction 505 of FIG. 5 may be as follows:

CC, DST = **OPER1** SRC1, SRC2; $HANDLER1

The handler label, if any, appears after the semi-colon. When the instruction does not throw an exception, the handler field is set to NULL. This may be either specified by the semantics of the instruction or found to be the case as a result of optimization or program analysis. In that case, the instruction may be textually denoted as follows:

CC, DST2 = **OPER1** SRC1, SRC2;

In cases where there is no destination operand or result for an instruction the destination and the "=" sign in the instruction description is omitted. For example, a conditional branch instruction does not have any explicit destination operands and its may be represented textually as follows:

CBRANCH SRC1, SRC1-LABEL, SRC2-LABEL;

Exception Handling Instructions

The following paragraphs describe the various exception handling related instructions of the intermediate representation by describing their operations, their inputs and outputs. Examples will illustrate how this instruction set can be used to generate an intermediate representation of exception handling constructs of various models within the same control flow as those instructions that are unrelated to exception handling.

Unwind

UNWIND	Propagate control out of the current method
Syntax	**UNWIND** x

Table 1

An UNWIND instruction is used to represent control flow out of the current method when no matching handler for an exception is present. The unwind instruction is preceded by a label, and is followed by an exit out of the method. The source operand (x) of the UNWIND operation represents the thrown exception object. This makes the data flow explicit. There can be one or more unwind instruction in a method. However, having just one UNWIND per method allows for savings in intermediate representation space for each method. Also the handler field of an UNWIND instruction is usually set to be NULL.

FIGS. 6 and 7 illustrate the use of an UNWIND instruction in the intermediate representation. FIG. 6 shows the pseudo code for an unguarded region that may cause an exception. In the process of translating to the intermediate representation, the IL 220 reader will be parsing the code in an intermediate language (210-213) representation and not the pseudo code. However, the pseudo code is being used in these examples in order to simplify the illustration of the control flow. For example, in FIG. 6 the

expression x = a div b may cause an exception if a divide by zero operation is attempted. Even if the original source code or its intermediate language representation (e.g., in MSIL or CIL) fails to specify a handler for this region the intermediate representation may provide a default handler which is usually an UNWIND instruction. Thus, an intermediate representation for the code of FIG. 6 may be as shown in FIG. 7. In the intermediate representation the exception causing instructions are shown with their handler fields 710 filled out and pointing to the handler with a label $HANDLER which marks the beginning of the UNWIND instruction. Now if an exception is caused the UNWIND instruction will move the control flow out of the method.

Finalization

The control flow to and out of a finally region may be represented in the intermediate representation by a set of instructions that are related, e.g., FINAL, FINALLY and ENDFINALLY. The FINAL instruction in general handles the explicit transfer of control to a finally region, whereas the FINALLY instruction can accept transfer from a FINAL instruction or through an exception causing instruction with a handler. The ENDFINALLY instruction represents the control flow out of a finally region.

FINAL	Branch to the start of a finally region
Syntax	**FINAL** *Label, Continuation*

Table 2

A FINAL instruction represents an explicit transfer of control to the start of a finally instruction. The first source operand of this instruction is the start label of the associated finally instruction, and the second operand is the continuation label where control is transferred after the finally region is executed. The handler field of a FINAL instruction is usually set to be NULL.

FINALLY	Accept control transfer from a final or exception handling instruction
Syntax	E, R = **FINALLY**

Table 3

A FINALLY instruction has two destination operands. The first operand is the exception variable. This models the data flow of the exception object. The second operand is a label or a code operand for the continuation that is captured. When a FINALLY instruction is executed as a result of an exception the captured continuation is the label of the lexically enclosing handler, FINALLY label or UNWIND instruction. This continuation label is reflected as the handler field of the matching ENDFINALLY (see below). The handler field of a FINALLY instruction is usually set to NULL.

ENDFINALLY	Leave the finally region and branch to the continuation or unwind
Syntax	**ENDFINALLY** *E, R, [case-list] ; $HANDLER*

Table 4

An ENDFINALLY instruction has two or more operands. The first operand is the exception variable. The second operand is the continuation variable whose type is the type of a label or a code operand. It also has a case list that is used to represent possible control transfers for explicit final invocations in the program. An ENDFINALLY instruction must have its handler field set to the label of the lexically enclosing outer finally or handler (i.e., a FILTER or UNWIND instruction). If there is no exceptional control flow to the matching finally instruction then the handler field may be NULL. Furthermore, the destination operands E and R of the FINALLY instruction is the same as the source operands E and R of the ENDFINALLY instruction. This ensures data dependence between the two instructions which can be used by the back end components during code optimization.

FIGS. 8 and 9 illustrate an example of implementing a finally block from the IL representation to the intermediate representation using the FINAL, and FINALLY and ENDFINALLY instructions. FIG. 8 illustrates the pseudo code of a try block. No handler is specified in the source code or its intermediate language representation. However, unlike the previous example no default handler needs to be specified because the expressions 810 are not exception causing instructions. Thus, control flows to the finally region only explicitly. The intermediate representation for the code of FIG. 8 may be expressed as shown in FIG. 9. The expressions 910 do not have handlers specified. The FINAL instruction 915 explicitly transfers control to the finally region indicated by the label $FINALIZE which points to the finally block 920. Once the finally block is executed, control transfers to the continuation label indicated in the FINAL instruction 915 which in this case is "$END."

FIGS. 10 and 11 illustrate translation of yet another try-finally block to an intermediate representation with FINAL, FINALLY and ENDFINALLY instructions. However, in this representation handlers are added to exception handling instructions. FIG. 10 shows the instructions 1010 which may cause exceptions that are guarded by the finally block 1015. FIG. 11 illustrates the intermediate representation of the try finally block along with exception handlers. The exception causing instructions at 1110 are assigned a handler label $FINALIZE directed to the beginning of the finally instruction at 1115. In this example, two types of control flows through the finally region is modeled. First, the FINALLY and the ENDFINALLY instructions are executed through an exception causing operation at 1110. In that event, after the ENDFINALLY instruction 1120 is executed, the control is passed to the region marked by the $PROPAGATE label. This in effect captures the continuation on the exception path. However, control flow to the finally region can also be transferred explicitly through the FINAL instruction 1112. In that event, at the end of the ENDFINALLY instruction 1120 the continuation is to the region marked by the $END region which does not implement the UNWIND instruction.

Yet another set of exception handling intermediate representation instructions for representing a finalization control flow may be referred to as the FAULT and the ENDFAULT instructions. They are similar to the FINALLY and the ENDFINALLY instructions, however, unlike a FINALLY instruction control flow cannot be passed explicitly from FINAL instruction to a FAULT instruction. Control to the FAULT instruction is branched to only through an exception causing instruction.

FAULT	Handle a fault.
Syntax	E = **FAULT**

Table 5

The ENDFAULT instruction terminates a related FAULT handler and throws the exception to a specified handler. The handler field can be NULL if all exceptional control flow to the corresponding FAULT instruction has been removed. In that case the fault handler is unreachable and can be deleted.

ENDFAULT	Leave the fault region/handler and search for an exception handler.
Syntax	**ENDFAULT** E; $HANDLER

Table 6

Filter based handlers

Some intermediate languages (e.g., MSIL) implement a filter-based handler, whereby different handlers are assigned to exception causing events based on a characteristic of the exception causing event. Such control flow may be represented in a intermediate representation using instructions to catch and filter exceptions and then to specify handlers to exceptions (e.g., FILTER, ENDFILTER and TYPEFILTER). As described below, TYPEFILTER instructions may be a short form of the FILTER and ENDFILTER instructions.

FILTER	Catch and filter an exception.
Syntax	E = **FILTER**

Table 7

This instruction may be used to implement a general-purpose filter-based handler in MSIL. This matches any exception and simply returns the exception object in the destination operand of the instruction. The filter instruction is labeled, and is followed by an arbitrary sequence of instructions that may or may not use the exception variable. A filter instruction must eventually reach an ENDFILTER instruction without an intervening FILTER instruction. The handler field of a FILTER instruction is usually NULL.

ENDFILTER	Terminate a non-resumption filter.
Syntax	**ENDFILTER** *X, handler-label, filter-or-unwind-label*

Table 8

An ENDFILTER instruction tests a Boolean operand (X) and if it is 1, branches to the handler label otherwise it tries another filter or unwinds.

TYPEFILTER	Catch an exception of the given type.
Syntax	E = **TYPEFILTER** *handler-label, filter-or-unwind-label*

Table 9

A TYPEFILTER instruction tests if the type of the exception object is a subtype of the type of the destination operand (which is statically known). If so, control is transferred to the first label (the handler label). Otherwise, another filter or unwind instruction label is tried. When the type filter matches, the destination operand is set to the exception object. The handler field of a TYPEFILTER instruction is usually NULL.

Note that a TYPEFILTER instruction is a short form and in fact can be represented as a combination of both FILTER and ENDFILTER operations as follows:

```
t.obj32 = FILTER;
e.Type  = CHKTYPE t.obj32;
x.cc    = CMP(NEQ) e.Type, 0.null;
ENDFILTER x.cc, $LABEL1, $LABEL2;
```

The FILTER instruction returns an exception object whose type is verified to be of e.Type and if it is of e.Type then, x.cc is set to TRUE and to FALSE otherwise. Then at ENDFILTER, the continuation is determined to be $LABEL1 or $LABEL2 depending on the value of x.cc. The same expression can be represented as a TYPEFILTER instruction as follows.

```
e.Type = TYPEFILTER $LABEL1, $LABEL2;
```

FIGS. 12 and 13 illustrate the implementation of a try–catch block using the filter based exception handling instructions. FIG. 12 describes an exception causing try region 1210 being guarded by two different handler regions 1220 and 1230. Filters at 1215 and 1225 determine which handler to implement based on the type of the exception object returned. FIG. 13 illustrates an intermediate representation of the try-catch pairs using the TYPEFILTER instruction of the intermediate representation. The handler fields of both the exception causing instructions 1310 is set to $HANDLER1 which points to the first filter 1315. If the exception object is of the type DivideByZeroException, then control flows to the catch block labeled $CATCH1. If not then the control flows to the next filter 1325 referenced by the label $HANDLER2. Based on whether the type of the exception object is type Exception, control flows either to the second catch block 1330 identified by the label $CATCH2 or to UNWIND instruction 1335 identified by the label $PROPAGATE.

A MATCHANYFILTER instruction is a form of the filter based exception handling instruction.

MATCHANYFILTER	Match any exception type.
Syntax	E = **MATCHANYFILTER** *handler-label*

Table 10

This filter always matches any exception unconditionally, and transfers control to a valid label. This is equivalent to a FILTER-ENDFILTER pair where the first operand of the ENDFILTER is always 1 and the second label is not specified. The handler field of a MATCHANYFILTER instruction must be NULL. A MATCHANYFILTER instruction is also a short form and can be represented using FILTER and ENDFILTER instructions as shown below.

```
e.Type = FILTER;
ENDFILTER 1.cc, $LABEL1, $LABEL2;
```

The equivalent MATCHANYFILTER instruction for the FILTER and ENDFILTER combination above is as follows:

```
e.Type = MATCHANYFILTER $LABEL1;
```

Representing a try block guarded by a filter-based handler and a finalization

Yet another exception handling model has the control flowing from a try block to one or more handler regions based on type of exception caused and then one or more finally regions. FIGS. 14 and 15 illustrate one such example. FIG. 14 shows the pseudo code for the intermediate language representation (e.g., MSIL) of a try block 1410 guarded by a pair of handlers 1420 and 1430 and a finally block 1440. Filters 1415 and 1425 determine which of the two handler blocks are processed based on the

type of the exception object returned. Regardless of the catch block traversed, the finally block will have to be touched before exiting the method.

FIG. 15 illustrates an intermediate representation of the control flow shown in FIG. 14 using the filter-based TYPFILTER instruction and the finalization instructions of FINAL, FINALLY and ENDFINALLY. This example exhibits several interesting points that do not occur in the previous examples. First, each handler 1520 and 1530 is terminated by an explicit invocation of the finally instruction at 1521 and 1531 respectively. This reflects the semantics of the original program as shown in FIG. 14 where control flows from the handlers 1420 and 1430 to the finally block 1440. Second, the filter 1525 that controls the last handler 1530 specifies the label for the finally instruction and not the unwind instruction 1550. This ensures that if the last filter is not matched it transfers control to the proper finally block. This point will be further illustrated in the examples for the nested cases of exception handling as well.

FIGS. 16 and 17 illustrate nested exception handling. FIG. 16 illustrates the pseudo code for the intermediate language representation of a try-catch-finally block nested within a try part of the try-catch block. FIG. 17 illustrates the intermediate representation of such a exception control flow using the filter, handler and finalization instructions described above. As shown in FIG. 16, the outer try block 1610 is guarded by two handler blocks with filters at 1620 and 1630. The nested try block 1615 is guarded by a catch block 1625 with a filter and a finally block at 1635. There are several exception paths possible here based on several factors including where within the source code the exception occurs. All these various exception paths can be expressed using the intermediate representation exception instructions as shown in FIG. 17. The exception causing instruction 1705 is guarded by the outer filter-based handler block labeled $HANDLER1 1710 which may pass control on to yet another filter-based handler block labeled $HANDLER2 1715. The exception causing instruction 1706 that is within the inner try block 1615 is guarded not only by the inner filter-based handler block labeled $HANDLER3 1720 but its exception path may also

pass through the block labeled $HANDLER1 at 1710 and/or $HANDLER2 at 1715. For example, if a DivideByZero exception is caused at the expression within the inner try block at 1706 then the exception path reaches the appropriate handler block at 1710 though the finalization block at 1725 by setting the handler field of the ENDFINALLY block at 1726 to the $HANDLER1 label. This flow represents the flow from try block 1615 to finally block 1635 and then to handler 1620.

A method for translating exception handling constructs from intermediate language code to a lower-level intermediate representation

As shown in FIG. 2, the intermediate representation of the exception handling construct using the instructions described above may be generated by an IL reader 220 which processes code in a intermediate language to generate such a representation. A IL reader 220 may use any number of different processes or algorithms and the choice or design of the algorithms may be dependent in part on the intermediate language itself and more particularly its model for representing exception handling constructs. For example, FIG. 4 shows an IL reader 445 suited for reading the source code representation in the MSIL intermediate language and generating a intermediate representation using the exception handling instructions described above.

FIG. 18 illustrates one possible method for generating an intermediate representation of exception handling instructions. In some intermediate languages (e.g., MSIL) exception data may be captured within a data structure separate from the main code stream. The IL reader (translator) for such languages may receive as input an exception handling data table 1810 and the main code stream in the intermediate language form 1820. Such data may then be read by an IL reader at 1830 to determine the relationship between exception causing instructions and any catch, finally or filter blocks associated with such instructions to generate the intermediate representation at 1840.

For example, FIG. 19A illustrates a data table containing exception handling data that may be available as part of an intermediate language representation of source code. The data shown in FIG. 19A corresponds to the code segment illustrated in FIGS. 12 and 13. As noted above, for sake of simplicity, FIG. 12 only shows the pseudo code of the intermediate language representation. However, the IL reader will in fact be parsing or reading the intermediate language code such as MSIL. The offsets 1240 to various instructions may be noted as shown in FIG. 12. The IL reader will be aware of the offset pairs enclosing distinguishable blocks of code. For example, the exception handling data of FIG. 19A notes the offset entries for the try block 1210 at 1910, a type of block that it is (e.g., try-catch, try-finally, try-catch-finally etc.) at 1915 and the offset entries for its handler blocks at 1920. Similarly, the offsets entries for the finalization blocks, filter blocks, continuation blocks and other blocks on the exception paths of instructions and their relationships to each other may be noted in form of offset entries as shown in FIG. 19A.

However, as noted above, the exception instructions of intermediate representation uses labels to mark or identify various cohesive blocks of code and these labels are used to build the intermediate representation along with the control flow. Thus, IL reader uses the exception handling data tables such as the one shown in FIG. 19A to generate the labels for building the intermediate representation. FIG. 20 is one example of a method for processing input in the form of an intermediate language code including captured exception handling data which is used to generate an intermediate representation. At 2010, if there are no more methods to be parsed or read within the code the translation process is stopped at 2015. If not, the current method and its associated exception handling data table are read at 2020. At 2030, the exception handling data including the offset ranges for each block of code is used to establish a containment relationship between the blocks of code that may be protected and the blocks of code that form its handlers, filters, finalization, continuations blocks etc.

For example, FIG. 21 illustrates building a tree data structure to map the containment relationship between the various blocks of code shown in FIG. 12. First, the offset belonging to the entire method 2110 is assigned a node on the tree. Then as each offset range in the exception handling data (FIG. 19A) is read they are assigned other relationships such as which ranges are guarded by which handlers based on the information provided by the exception handling data. For example, Try-Catch offset range 2120 is shown as containing the Try range 2130 that is guarded by two handlers the first Catch at 2140 and a second Catch at 2150. This example shows a single case of a try-catch block within a method but the tree structure can get much larger as multiple blocks of code are nested within each other and the tree structure provides a suitable way to represent such multiple containment relationships.

Returning now to FIG. 20, as the containment relationship between various blocks of code is established (2030), at 2040, each of the handler and destination blocks (e.g., finally or continuation, etc.) which may be identified only by their offset ranges can now be assigned distinctive labels. For example, FIG. 19B illustrates the labels $HANDLER1 and $HANDLER2 being assigned to the two handler blocks identified in the code of FIG. 12. Returning again to FIG. 20, at 2050, using the containment relationship of the various blocks of code the protected blocks are assigned labels to their handlers and destination blocks as shown in FIG. 19C. This allows the expression of the relationship between the protected blocks and their associated handlers and destination block in form of labels in the intermediate representation expressions described above. Once the protected blocks are mapped to their appropriate handlers and other destination blocks, at 2060, the intermediate representation for the method is built by parsing the code again. In this pass, each instruction is read, and if it is an exception causing instruction, its handler is identified using the range tree data structure built earlier. This process may be repeated until all the methods within a program are translated to their intermediate representation.

Intermediate representation of object construction and destruction as try-finally blocks

Some languages such as C++ allow a programmer to declare object (class) variables that have local lifetimes within blocks or expressions where they are declared. The language semantics requires that the corresponding object destructors be called when the block scope or the expression scope is exited. These operations can be represented within the intermediate representation as one or more sets of try-finally blocks. For example, FIG. 22 illustrates a program using local objects. Statements S1 and S3 generate a call to the constructors for class1 and class2, respectively. The constructors for these classes and the statements S2 or S4 calling on these objects might throw an exception. In that event, appropriate cleanup might be necessary. If statement S1 throws an exception, the exception is propagated because no object has yet been successfully created. If S2 or S3 throws an exception, the destructor for class1 needs to be called. However, if S4 throws an exception the destructor for class2 needs to be called followed by the call to the destructor for class1. Since this example does not have a handler, the exception is propagated to the caller of the method. These operations may be expressed, conceptually, by the nested try-finally construct shown in FIG. 23. The constructor for obj1 2310 is outside any of the try blocks. Thus, if an exception is thrown during the constructor 2310 no objects need be destructed prior to exiting the method. However, if obj1 is successfully constructed at 2310 then it has to be destructed passing through the finally block at 2320. However, if control reaches the inner try block at 2330 then both obj1 and obj2 have to be destructed by passing through both the finally blocks 2320 and 2340. The intermediate representation using the FINAL, FINALLY and ENDFINALLY instructions for these operations may be as shown in FIG. 24. The two FINAL instructions 2410 and 2420 provide the explicit entry to two sets of FINALLY and ENDFINALLY instructions represented at 2430 and 2440 respectively. Control flow may reach these instructions through exceptions as well in the event an exception is caused during destruction of the second object. The

destructor instruction for the second object at 2435 has a handler label ($DTOR1) pointing to the FINALLY and ENDFINALLY instructions 2440 containing the destructor for the first object at 2445. This is so because if control flow has reached the destructor 2430 of the second object, then necessarily the first object must have been constructed and so has to be destroyed before the method is exited. Even if there are no exceptions thrown when destructing the second object, the first object will still be destructed at 2440 ($DTOR1) prior to exiting the method.

Intermediate representation of expression temporary objects

Some languages such as C++ allow creation of expression temporary objects. These objects are created during expression evaluation and are destroyed after the expression evaluation, typically after the evaluation of the statement containing the expression. If the expression is a conditional expression, then objects created have to be destructed conditionally. For example, FIG. 25 shows expression temporaries obj1(x) and obj2(x+1). The destructors for these objects have to be called after the call to foo(). However, note that the creation of these objects occurs conditionally depending on the value of the variable "x." Thus, the destructor instructions for these objects also have to be guarded by the same condition. This may be expressed as a set of nested try-finally blocks as shown in the pseudo code of FIG. 26. Depending on the value of "x" obj1 or obj2 is created at 2610 and 2620 respectively. Thus, based on the same condition obj1 or obj2 have to be destructed at 2630 and 2640 respectively. Unlike the previous example, here only one object is created at any given time. FIG. 27 illustrates one intermediate representation for this construct using multiple sets of FINAL, FINALLY and ENDFINALLY instructions described above. Depending on the value of "x", a branch instruction 2710 points to the code for the constructor of obj1 at 2720 or to the code for the constructor of obj2 at 2730. Also note that if an exception is thrown during the creation of either of the objects at 2720 or 2730 the handler label is set to $PROPAGATE (2721 and 2731) which marks a UNWIND instruction to pass the

control outside of the method. Then again depending on the object created, either obj1 is destructed at 2740 or obj2 is destructed at 2750 both such destructors are contained within a pair of FINALLY and ENDFINALLY instructions. In this manner, conditional creation and destruction of expression temporary objects may be represented using intermediate representation instructions.

Intermediate representation of the returning of objects by value

Some languages such as C++ permit returning objects by value. Before returning the object, destructors are called on the locally created objects. For example, in FIG. 28 objects a and b are locally created at 2810. If the destructors on any of these local objects throws an exception, then the return objects r1 at 2820 and r2 at 2830 should be destroyed before exiting the method. The exception handling control flow for the code in FIG. 28 may be expressed in the intermediate representation as shown in FIGS. 29A and 29B. Before returning the object r1 at 2910, the method has to call the destructors of locally created objects a, and b at 2920 and 2930, respectively. However, if any of these destructors 2920 or 2930 throw an exception, then the destructor of the return object 2940 has to be called. Such a control flow may be represented as shown in FIG. 29 using the appropriate sets of FINAL, FINALLY and END FINALLY instructions. For example, the handler label ($final_a1) of the destructor of object b at 2930 points to label for the destructor of object a 2920 whose handler label ($final_r1) in turn points to destructor of the return object r1 at 2940. This ensures that if there is an exception caused during destruction of objects a or b or both, the return object r1 is also destructed before exiting the method. Note that at 2940, the destructor for r1 is called if the flag $F1 is equal to "1", which is set to "1" at 2950 and remains set to "1" if the either object a or object b could not be successfully destroyed. In this manner, return object r1 is destructed if destruction of object a or b is unsuccessful. The conditional destruction of the return object r2 is handled in the same manner.

Intermediate representation of the throwing of objects by value

Some languages such as C++ permits throwing and catching objects by value, i.e., values allocated on the stack. Simple primitive values like type "int" do not pose any issues. However, throwing structs and classes allocated on the stack may require calls to constructors and destructors. FIG. 30 illustrates source code for throwing an object by value. Conceptually, the operation may be represented in the pseudo code as shown in FIG. 31. In this example, a copy of the local value is made and the copy constructor 3110 is called on the copy. When the value is thrown at 3120 the destructor for the new copy of local value has to be passed so that method receiving the value can call the destructor at a later time. A finally block 3130 guards all exceptions and is responsible for calling destructors.

The try-finally block may be represented in the intermediate representation as a set of FINAL, FINALLY and ENDFINALLY instructions and following instruction may be used to represent throwing of value types which have copy constructors and destructors defined for them within the copy instruction.

THROWVAL	Throws a value type as an exception
Syntax	**THROWVAL** *E, Dtor* ; $HANDLER

Table 11

This is a special form of throw that is used to throw value types which have copy constructors or destructors defined for them. It has two operands. The first operand is a pointer to the location that has the value being thrown, and the second operand is a function pointer that performs the destruction. The semantics of this is that the thrown object is destructed when a handler is found. This essentially keeps the local value type location live at runtime. This may be used to model the C++ exception semantics for value types. The handler field of a THROW instruction is usually not set

to NULL. FIG. 32 illustrates an intermediate representation of the code of FIGS. 30 and 31. The THROWVAL instruction 3210 is used to represent value throwing and in this example, it is shown receiving the pointer to the location of the value being thrown 3220 and the pointer to its destructor 3230 to be used later by the methods receiving the thrown object.

Intermediate representation of a try –except construct

Structured Exception Handling (SEH) extensions to languages such as C and C++ provide an exception handling construct expressed as a try-except block. FIG. 33 illustrates a try-except block. Like a catch block with a filter, an except block specifies pointers to handlers of an exception causing instruction based on the type of exceptions caused. However, except blocks also allow for the possibility of resuming the execution of an instruction that caused the exception. The following two intermediate representation expressions may be used to represent such an exception control flow.

SEHENTER	Enter an SEH guarded region.
Syntax	**SEHENTER** $HANDLER

Table 12

An SEHENTER instruction marks an entry to a try-except region. Its handler specifies a control dependency to the handler and the body of the guarded region.

ENDRESUMEFILTER	Terminates a resumption filter.
Syntax	**ENDRESUMEFILTER** *X, handler-label, filter-or-unwind-label, resume-label*

Table 13

An ENDRESUMEFILTER is similar to an ENDFILTER except that it may cause the execution of the exception causing instruction to be resumed when the source

operand has a value of -1. FIG. 34 illustrates an intermediate representation for the try-except construct of FIG. 33 using the SEHENTER, FILTER and ENDRESUMEFILTER expressions described above. At 3410 the SEHENTER instruction is used to prefix the call to the body of try 3420. Also, to ensure proper control dependencies among the operations in the try-except region the handler to the SEHENTER expression is set to the FILTER instruction 3430 as is the call to the exception causing instruction foo() at 3420. The ENDRESUME instruction is used to denote the continuations of the exception path based on a value returned by the filter function 3450. If the value "t" is 1, then the control passes to the handlerbody ($HANDLERBODY). If the value is "0", then the method is exited. However, if the value of t returned is "-1" then control returned to $LABEL to resume the execution of the operation that caused exception in the first place. Also, the representation of FIG. 34 provides one exception path, which has an exit directly from the SEHENTER instruction 3410. This ensures that only safe code-motion is done. In the same representation, however, the call to the filter function at 3450 does not have a handler but a handler such as an unwind instruction (not shown) may be set if the filter() function is likely to cause an exception.

An alternative method for translating an intermediate language representation of exception handling constructs from intermediate language code to a lower-level intermediate representation

As noted in FIG. 4, a separate IL reader may be necessary for generating the intermediate representation from different intermediate languages (e.g., CIL, MSIL etc.) in order to conform to the exception handling models specific to that particular language. The following section describes a method for generating an intermediate representation of exception handling constructs from a language that expresses operations in form of post fix notation as does CIL.

Generally, in post-fix notation expressions, the operands of an operation are expressed before the operator is expressed. For example, in the code for an ADD

operation such as T = 5 + 3, a reader will encounter the code for the operands 5 and 3 before encountering the code for the operator "+" (i.e., ADD). A translator for such code, which uses a post-fix notation form and more particularly, one capable of translating such code in one pass may translate the code for the operands first, and then build the translated code for the entire operation based on the code for its operands or its children nodes (also referred to as sub-expressions of an expression).

FIG. 35 illustrates one overall method for generating an intermediate representation of exception handling constructs from an intermediate language that uses post fix notation such as CIL. At 3510, the input code is read node by node (i.e., expression by expression). Then at 3520, because of the recursive nature of the post fix notation form a context for current node within the rest of the code stream is determined. This context can later be used at 3530 to put together the code for parent nodes based on the code for their children nodes.

Post-fix notation language may express exception information in form of operations using selected operators, which may be processed by an IL reader in the manner of FIG. 35 to generate the intermediate representation in form of the uniform framework of instructions (e.g., FINAL, FINALLY, and ENDFINALLY). FIG. 36 illustrates one way of implementing the method of FIG. 34. FIG. 36 shows several data structures (e.g., 3610, 3620, 3630) to serve as building blocks for containing the translated intermediate representation code which can later be combined to together complete the intermediate representation form for a code section such as a method. The data structures 3610 and 3630 may be implemented as conceptual stacks with their own nodes having their own data structures. As the code in the intermediate language form (e.g., CIL) is read and translated the intermediate representation code related to each sub-operation, sub-expression, child node etc. may be stored within the data structures (e.g., 3610, 3620, 3630). Then as the context or containment relationship for each operation is established or at other appropriate times all the translated intermediate code is added together to generate the complete translation.

In one such method, nodes read from intermediate language input are pushed on to the evaluation stack 3610 where they may be evaluated. The evaluation of a node read from the input may require popping of some nodes from the evaluation stack or EH stack, and pushing new nodes onto the evaluation stack or EH stack. Generally, the evaluation stack 3610 may contain intermediate code related to most of the main code stream. Some nodes from the evaluation stack may be then popped off the stack to build code for other nodes as the context or containment relationship of the parent and children nodes are established. For example, getting back to the simple add expression T = 5+3, when its established that 5 and 3 are operands for the "+" operation, the nodes on the evaluation stack 3610 related to the constants 5 and 3 are popped and the code for the add operation can be synthesized by composing the code of its children, namely nodes representing 5 and 3. The translation algorithm uses and maintains the invariant that nodes on evaluation stack have all attributes computed.

The DTOR code data structure 3620 may be thought of as a structure for encapsulating the translated intermediate representation code sequences for all object destructors, catch blocks, and finally blocks that appear in the body of a method. The exception handling (EH) stack 3630 is not used to contain code sequences, as such, but may be thought of as a stack of continuations used to establish the relationship between the various sequences of code by building labels. The EH stack establishes the nesting relationship among try, catch and finally regions. Each region can be identified by the label associated with the region. Each node in the EH Stack has a unique ID called the state. The notion of state is used to compute the number of objects allocated in an expression evaluation. This information can be used later for such things as determining the number of destructors that need to be added to the translated code and their relationships to the rest of the code.

The data structure for each of the nodes on the evaluation stack 3610 may be as shown below.

Field	Description
Opcode	IL opcode of the node
IL Type	The IL Type of the node as provided by the front end of the compiler
Opnd	IR data structure representing the Opnd that is the result of evaluation of the node. This field is empty when the node is read and the evaluation of the node computes the Opnd field.
EHStart	Integer representing the top of EH stack when node was read
EHEnd	Integer representing the top of the EH stack when the node is evaluated. The difference between EH Start and EH End numbers gives the nodes that were pushed onto EH Stack during the evaluation of the expression. It is used for generating Finally regions of expression temporaries of conditional expressions.
FirstInstr	Points to the beginning of the translated code sequence related to the node
LastInstr	Points the ending of the translated code sequence related to the node

Table 14

Only the "Opcode" and the "IL Type" fields may be provided by the front end of the compiler the rest of the fields are filled during the translation process as the intermediate representation code is generated. The FirstInstr and the LastInstr fields will allow the concatenation and pre-pending of code. The entire DTOR code data structure 3620 may be implemented similar to the data structure of one node of the evaluation stack 3610.

The EH stack nodes may have the following data structure for representing the continuations on the exception path.

Field	Description
Label	A label to a finally block or a catch block
Flags	Flags representing the EH flags
State Type	Represents state type
State Id	Represents the EH state id

Table 15

The label field points to a label instruction that can precede a Finally, or TypeFilter instruction. The flag field may be used to identify the characteristics of the exception handling operations being performed. For example, it may be used to identify whether destructor code to be translated is for a temporary object such as an expression temporary or whether it is for an object that can be destructed outside of the expressions within which it is constructed.

Within the context of the data structures for the nodes of the evaluation stack 3610, DTOR code 3620 and the EH stack 3630 a method for generating intermediate representation as shown in FIG. 35 may be described in further detail as shown in FIG. 37. The method of FIG. 37 is implemented for each method within the intermediate language stream being translated. At 3710, the translation data structures shown in FIG. 36 (e.g., 3610, 3620, and 3630) are initialized. For example, the evaluation stack 3610 and the EH stack 3620 are initialized to be empty. Also, an UNWIND node may be pushed on to the EH stack because all methods will at least have one exception path continuation to the outside of the method and the current state of the EH stack may also be initialized. Later at 3720 a node from the intermediate language code is read and its EH state is initialized to the CurrentEHState. Again, states are used to maintain the context of the code being read recursively. Later at 3740, the node is evaluated within the existing context of the evaluation stack and the EH stack. The result of evaluation is the definition of Opnd, FirstInstr and LastInstr fields of the node and a change in the context, (i.e., nodes can be pushed or popped of the evaluation stack or EH Stack and value of CurrentEHState can be modified). Nodes on the evaluation stack represent the evaluated nodes, which have their fields completely filled. Then at 3750, if the operation being evaluated is not an exit the next node is read. However, if it is the exit of the method then at 3760, the translated code contained within all the evaluated nodes on the evaluation stack 3610 are prepended and the code

within DTOR code data structure 3620 is concatenated to this result to yield the complete intermediate representation code.

An example translation of intermediate language code in post-fix notation to a lower level intermediate representation

The method of evaluation 3740 may be different for different operations as they are encountered during the process of reading the intermediate language code. The following example illustrates the method for evaluating some such operations in intermediate language code. FIGS. 22 and 23 illustrate the pseudo source code for object constructors and destructors in a language such as C++. FIG. 24 illustrates the code of FIGS. 22 and 23 translated to the intermediate representation using the instructions such as FINAL, FINALLY, ENDFINALLY. The post-fix notation intermediate language code being processed by the IL reader (i.e., translator) may be in the form shown in FIGS. 38A, 38B and 38C. Such code may be read and translated to the intermediate representation (FIG. 24) using the methods described above with reference to FIGS. 36 and 37.

FIG. 39 illustrates the process of translating the intermediate language code of FIGS. 38A, 38B and 38C containing object constructors and destructors using translation data structures (e.g., 3905, 3915, 3925). At first, all the data structures are initialized to be empty. Then as the code in FIGS. 38A, 38B and 38C is read each operation is evaluated and nodes are pushed on to the evaluation stack 3905 or popped to generate code within the DTOR code data structure 3925 and to manipulate the EH stack 3915. In this example, the constructor of object 1 is identified first at 3810, this is pushed as a node on to the evaluation stack at 3910. Later at 3820, the code for the destructor of object 1 is encountered which is also temporarily pushed on to the evaluation stack. When the reader encounters the Oppushstate operator at 3830 then it is known that the destructor 3820 and constructors 3810 were operands of the Opushstate operator 3830. It is also known that the destructor code is a handler so

needs to be placed within the DTOR code data structure 3925. Thus, the top node on evaluation stack 3905 related to destructor of object 1 is popped and eventually appended to the DTOR code data structure 3925 along with a label identifying the block. The DTOR code data structure 3925 is also appended with the FINALLY and ENFINALLY instructions with the appropriate continuations as shown in FIG. 39. The EH stack 3915 would have been initialized to have a node 3920 with a label preceding the UNWIND instruction, but now a new node 3940 is added to the EH stack and its label is set to the label added to the block of code containing FINALLY and ENDFINALLY instructions. Up to this point the intermediate code related to the exception paths and continuations have been determined. The code for object 2 is evaluated in a similar manner. Later at 3840 when the Opdtoraction operator is encountered the intermediate code related to explicit entry (i.e., FINAL instruction) into the finally region is built as shown at 2410 and 2420. FIG. 39 shows the state of the evaluation stack 3905, the DTOR code data structure 3925, and the EH stack 3915 after the intermediate language code of FIG. 38 upto the point where the OPpushstate instruction 3830 has been evaluated. Thus, all the code necessary for stitching together the code for intermediate representation is contained within the data structures 3905, and 3925 and their various nodes, which can be added to form the complete translated code. Although, the method for evaluating and stitching together code for the various operators may be different, the various data structures described above may be used within the context of each individual operator and its function and form to generate the desired intermediate representation.

Alternatives

Having described and illustrated the principles of our invention with reference to the illustrated embodiments, it will be recognized that the illustrated embodiments can be modified in arrangement and detail without departing from such principles. Although, the technology described herein have been illustrated via examples using

compilers, any of the technologies can use other software development tools (e.g., debuggers, optimizers, simulators and software analysis tools). Also, it should be understood that the programs, processes, or methods described herein are not related or limited to any particular type of computer apparatus. Various types of general purpose or specialized computer apparatus may be used with or perform operations in accordance with the teachings described herein. Actions described herein can be achieved by computer-readable media comprising computer-executable instructions for performing such actions. Elements of the illustrated embodiment shown in software may be implemented in hardware and vice versa. In view of the many possible embodiments to which the principles of our invention may be applied, it should be recognized that the detailed embodiments are illustrative only and should not be taken as limiting the scope of our invention. Rather, we claim as our invention all such embodiments as may come within the scope and spirit of the following claims and equivalents thereto.